http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

Information Technology Journal 9 (3): 576-582, 2010
ISSN 1812-5638
© 2010 Asian Network for Scientific Information

A Software-Implemented Fault Injector on Windows NT Platform

Qing-He Pan, Bing-Rong Hong and Qi-Shu Pan
School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China

Abstract: In this study, we present our experience in developing a tool for Software-Implement Fault Injection (SWIFI) into Windows operation system. The fault injector uses software-base strategies to emulate the effects of radiation-induced transients occurring in the system hardware components. The SWIFI tool called MOFI (Memory Oriented Fault Injector) is being used, in conjunction with an appropriate system fault model, to evaluate the applications, measure the injecting strength of injector and mean time to failure of injected applications and determine the sensitivity of applications to faults. The MOFI has been validated to inject faults into user-specified CPU registers and memory regions with many random distributions in location and uniform random distribution in time. The different random distributions chosen in location could produce different experiment results. The reason will be discussed in this study.

Key words: Software-implemented fault injection, single event effects, injecting strength, mean time to failure, sensitivity

INTRODUCTION

Now more and more Commercial-Off-The-Shelf (COTS) hardware and software components are employed in spacecrafts. A fault tolerance mechanism for satellite processing application had been implemented through redundant COTS components (McLoughlin *et al.*, 2003) and Remote Exploration and Experimentation (REE) project was launched to bring commercial cluster technology into space (Some and Ngo, 1999). In future more applications and projects like them will run in space to execute scientific explorations. Generally software and hardware of these COTS are not radiation hardened to protect them from radiation, for example Galactic Cosmic Ray (GCR) and Solar Proton will cause Single Event Effects (SEEs) (Some *et al.*, 2001) in registers and memory. In order to reduce influence of SEEs some fault-tolerant mechanisms were designed and implemented by both hardware and software.

Recently, software-implemented fault tolerance became focus of research. Oh *et al.* (2002) brought up control-follow checking. Torres-Pomales (2000) discussed single-version and multi-version software fault tolerance techniques in detail. The theory and practice of fault-tolerant high-performance matrix multiplication in REE Project of NASA's High Performance Computing and Communication Program was designed and implemented by Gunnels *et al.* (2001). Before a fault-tolerant system is deployed, it must be tested and validated to check if fault-tolerant mechanisms take effect. These tests are more important in systems composed of COTS since, there are no radiation-hardened chips used. Usually these tests are implemented by fault injection techniques (Some *et al.*, 2001). These techniques are basically divided into four categories:

- Hardware-implemented fault injection (Madeira *et al.*, 1994)
- Software-Implemented Fault Injection (SWIFI) (Han *et al.*, 1995)
- Simulation-implemented fault injection (Blanc *et al.*, 2002)
- Hybrid fault injection-a combination of the former three methods

Hardware-implemented fault injection techniques need special hardware and may produce permanent harm to object hardware. Simulation-implemented fault injection techniques are difficult to create models, more cost in computation and inaccurate. The SWIFI techniques are naturally simulations of hardware faults in order make the system behave as if real hardware faults had occurred. Compared to the other techniques SWIFI techniques have some special advantages: (1) no complexity models, (2) less development effort, (3) lower cost and (4) increased portability (Some *et al.*, 2001). There are quite a few SWIFI based fault injectors such as FIAT (Han *et al.*, 1995), FERRARI (Kanawati *et al.*, 1995), NFTAPE (Stott *et al.*, 2002), DOCTOR ORCHESTRA (Dawson *et al.*, 1996), BOND (Baldini *et al.*, 2000), Holodeck (http://www.securityinnovation.com/holodeck/index.shtml) and Xception (Carreira *et al.*, 1998). They run on Unix, Linux and WindowsNT platforms. In this study, MOFI is like BOND and Holodeck, since all of them run on WindowsNT platform. But there are basic differences of injecting method between them.

Corresponding Author: Qing-He Pan, School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China

BOND and Holodeck focus on injecting faults into API (Application Programming Interface) parameters and when these parameters pass through interfaces of layers and components of software, faults are injected. So both of them need to intercept API, modify parameters and continue to pass. This way obviously can only inject determined faults into special locations but can't inject fault into any positions of registers and memory. The MOFI is different from them. It can inject faults into registers and any location of memory. The difference comes from the aim of injection. In both BOND and Holodeck SWIFI techniques are aided approaches to normal software test. In MOFI the aim is to emulate SEEs occurring in registers and memory in order to evaluate reliability of software which will run in spacecraft and other ground systems that work in high radiation environments.

Xception is similar with MOFI in functions. Xception operates at the exception handler level and needs debugger mechanism of CPU. Both MOFI and Xception minimize intrusion into application. The MOFI doesn't depend on specific hardware architecture.

This study describes the theory of MOFI fault injector and the methods to compute reliability parameters and analysis the influence of different injecting locations distributions on injecting experiments results.

INJECTING METHOD

MOFI can inject faults into registers and memory when object applications are running. The MOFI can inject one-bit or multi-bits faults that emulate SEEs effectiveness into registers and memory with the same method. There are only limited registers in x86 architecture where our experiment platform Windowsxp runs. In one research detailed techniques of injecting faults into 75 registers of POWERPC were studied (Some *et al*., 2001). The registers in x86 are less than POWERPC so the work of injecting faults is easier. So, in this study, we mainly focus on injecting faults into memory and all computations and experiments are surrounding it. First it is necessary to study the application memory image when it is running as a process.

Process is put into memory according to various regions such as code region, data region, heap region, stack region and so on. When MOFI injects faults into application it just injects faults into regions of process of application. Figure 1 is a concept display of process memory image. This concept display corresponds to almost any popular operation system that supports multitask and virtual memory. The number on left is the address of memory cell. The range is from 0x00000000 to 0xFFFFFFFF. Theoretically MOFI can inject faults into all memory cells of process of an application.

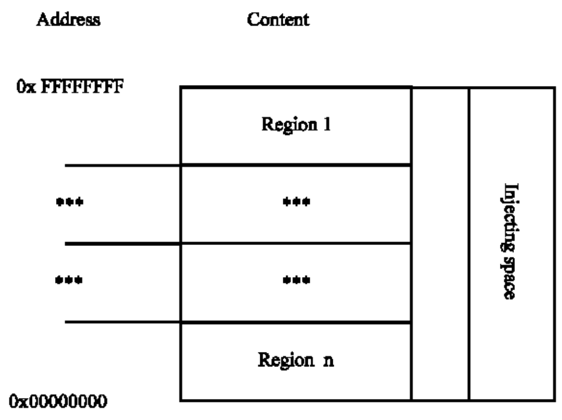

Fig. 1: Process memory image

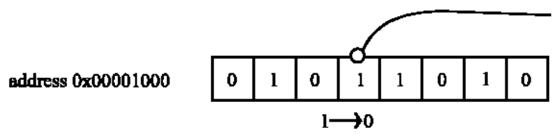

Fig. 2: SEEs illustration

Figure 2 shows what is the SEEs. The MOFI can emulate the SEEs effectively. In this example MOFI injects one-bit SEEs fault into address 0x00001000 by changing the fourth bit from 1 to 0. So after injected fault the content of address 0x00001000 becomes 01001010. The MOFI can inject one-bit or multi-bits fault into memory space of the running process and this is its main injecting method.

THEORY OF INJECTION OPERATION AND ALGORITHM

MOFI is an application-level software-implemented fault injector providing an easy-to-use environment for fault injection experiments including massive fault injection campaigns to get statistical data on fault performance and accurate fault injection into any determinate location for specific purpose. The user just needs to tell MOFI the name of object application that will be injected or PID (process identifier) of its process. Although, MOFI could inject faults into any applications, with or without source code, that run in windows, in our injecting experiments all applications are with source codes. This is important because if a campaign needs to implement 1000 injecting experiments and the injected applications are not silent with exceptions, the human operator might have to close windows Application Error dialog hundreds times. The massive human work is terrible and will affect the precision of time in experiments records. So, (SetErrorMode) MSDN, 2007. All API functions referred in this study come from it function could be added into source code and after compiled again the application will be silent with exceptions. With a configuration file and execution Tcl scripts, a campaign of a thousand fault injection experiments may be automatically run, results collected and evaluated and statistical results obtained without human interference. The injected application is to be created by injector using (CreateProcess) API. But after created the process of application is not as a subprocess of injector, there no longer exists relationship between application and injector.

Some *et al.* (2001) classified the result of each injecting experiment as Correct, Incorrect, Crash, Hang, or Invalid. The reason of so many classes is in that research faults were injected into code region, data region, heap region and stack region with uniform distribution. In present research the mainly injected region is code region and almost any fault injected into this region of a running application will cause it hang or crash. So the number of fault classes declines and judgment standard is easier: Injector observes injected application after a fault was injected into its memory space. The application is checked through (FindWindow) on a prescribed time value. If (FindWindow) return NULL within the time value it indicates that this injection makes application failure. Otherwise if (FindWindow) still could find window handler of application it indicates that this injection didn't make application failure. If (SetErrorMode) is used when windows operation system finds a failure occurring the most usual disposal route is to close the application and collect all resources allocated to it. The behaviors can be seen is the windows (or dialogs) opened by application disappearing or the PID of application disappearing in task management. So far the detailed techniques were discussed. Now an injecting method algorithm will be given.

Injecting Method Algorithm (IMA): Set N is the number of injecting experiments and T is a prescribed time value that injector needs to check whether the injected application is failure. In each experiment only a fault will be injected into an object application. After injecting a fault the injector will observe the injected application within T. If within T the injector find application failure then it finishes this experiment, records experiments results and begins next experiment (if number of experiment is less than or equal to N). If within T application doesn't failure the injector will consider that this injection can't hit application and begin next injecting experiment. Set application failure number is fn during all experiments are executed.

- **Step 1:** (Initiate) Set N, T and produce a list X(N) that contains all locations that will be injected. Each experiment chooses a location to inject. Set I = 0, fn = 0, m = 1
- **Step 2:** (Inject) If I = N, inject fault at location X(i), go to step 3. Else finish experiments and exit

Table 1: Experiment record format of IMA

#Num	Name	#Num	Name
#1	Inject-id	#7	Inject-cont
#2	Start-time	#8	Finish-time
#3	Inject-addr	#9	Execute-time
#4	Read- flag	#10	Failure-flag
#5	Read-cont	#11	Accum-fail
#6	Inject-flag		

- **Step 3:** (Observe) If within T application failure then fn = fn+1 and go to step 4. Else also go to step 4
- **Step 4:** (Record) Record the results of this experiment, I = i+1, go to step 2

In step 1, the way to choose X(N) is deserved to discuss. Different ways make different experiment results. This will be discussed at end of the study.

In Step 4, we need to record the results of experiments. The format of record is described by Table 1.

Table 1 gives all fields of one record. By algorithm IMA each experiment has one record so after N experiments N records will be produced.

#Num stands for number of data field. The description after it stands for name of data field. #1 records the injecting experiment id, for example for the ith (i = N) experiment the id is i. #2 records the starting time of each experiment. #3 records the location that will be injected in this experiment. #4 is a flag it indicates whether the content of location could be read. If content can't be read the Read-flag will be set to 0, else 1. #5 will store the content that is read. If #4'value is 0 then #5 will be set to FFFFFFFF. #6 and #7 are similar to #4 and #5. #6 indicates whether injector can inject fault into the memory cell pointed by #3. #7 stores the fault that is injected. #8 and #9 stand for finishing time of experiment and executing time, respectively. And #9 is also the surviving time of application after injected. #10 is a failure flag. It indicates whether this injecting experiment causes application failure. #11 stores accumulating number of failures from the first experiment. It equals value of fn in IMA.

In algorithm IMA the experiment time is mainly consumed in Step 3 and the value is O (1). So, all time needed to execute N injecting experiments can be computed as:

$$\Phi_{IMA}(N) = O(1) + \Phi_{IMA}(N-1) \qquad (1)$$

So, the result isΦ_{IMA} and it easy to know that max (Φ_{IMA}(N)) = TN.

Next section the method to compute interesting parameters by experiment results will be discussed.

ANALYSIS AND COMPUTATION OF PARAMETERS

Interesting parameters can be computed by the records of experiment results. To demonstrate how to compute parameters by format of record, a fault injecting campaign will be run. Injecting strength, MTTF (Mean Time To Failure) and fault sensitivity are chosen as parameters that will be computed. Any application can be used as injected object. In this campaign a GUI sine wave generator is chosen. The campaign will be executed by IMA. 50 experiments will be run so N = 50 and by scale of application code region the prescribed time value is set to 2 sec, T = 2 sec. For each parameter first we give its definition and computing formula by format of record then illustrate the computing results and analyze its meaning.

Definition 1: (injecting strength λ): It is the average value of number of application failures caused by injector per unit time. It can be computed by Eq. 2. All items on right of Eq. 2 are described in Table 1.

$$\lambda(j,n) = \frac{\text{accum\_um}(j+n-1) - \text{accum\_num}(j)}{\text{finish\_time}(j+n-1) - \text{start\_time}(j)} \tag{2}$$

where, λ (j, n) represents the injecting strength of injecting experiment form jth time to nth time. It is obvious that λ (j, n) is an average measure of injecting effect. We can extract #8 and #11 from experiment records and plot them in Fig. 3.

In Fig. 3, x-axis stands for time of the whole campaign. In this campaign 50 experiments are run and experiment time is about 120 sec. By Eq. 2 and Fig. 3, λ can be computed on any time interval between (0,120). If λ = 0 it means that injector didn't cause application failure during the corresponding time interval, for example during (0, 10) accumulated failure count is 0 so λ(0, 10) = 0.

Definition 2: (failure time): The time interval from time when application is injected to time when failure happens.

Usually average failure time, i.e., MTTF (Mean Time To Failure), is an interesting parameter. So by definition 2 we directly give the formula Eq. 3 to compute MTTF.

$$\text{MTTF} = \frac{\sum_{i=1}^{N}(\text{f\_flag}(i) \times \text{e\_time}(i))}{\sum_{i=1}^{N}\text{f\_flag}(i)} \tag{3}$$

where, f-flag (i) is #10 field in table 1 and e-time (i) is #9. The i in Eq. 3 is the sequence number of N experiments.

We can extract #1 and #9 from experiment records and plot them in Fig. 4.

In Fig. 4, x-axis stands for the number of experiments, i.e., N = 50. Y-axis is the application run time for each experiment. There are 37 experiments in

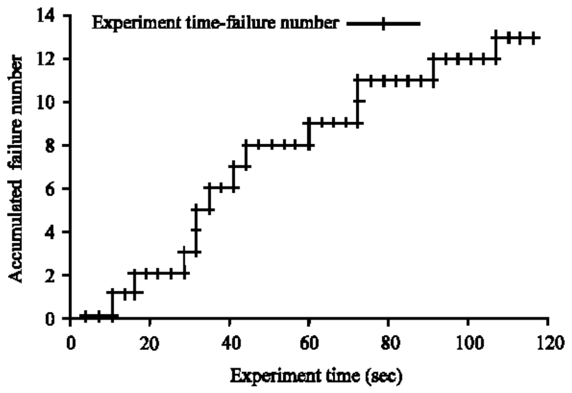

Fig. 3: Compute injecting strength

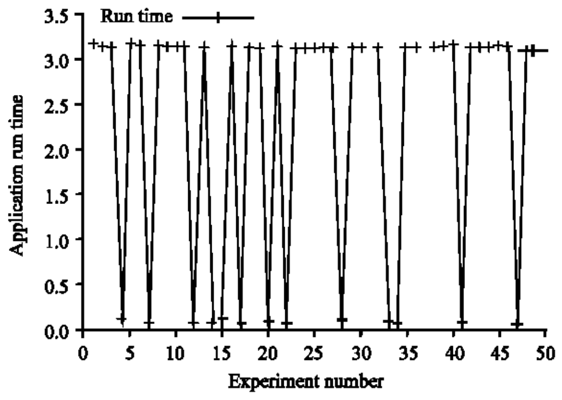

Fig. 4: Compute MTTF

which the run time is greater than 3.1sec and others run time about 0.2 sec. Because T = 2 sec and by IMA when t = 2 injector believes that this injection doesn't make application failure, in Fig. 4, we can see only 13 experiments are considered to make application failure. We can compute MTTF by experiment records through Eq. 3. In this campaign MTTF = 0.058 sec. This implies that when the sine wave application is running in space if SEEs happens in its code region it will fail in 0.058 sec. So, if deploying some fault-tolerant mechanism to avoid failure the mechanism should be effective in time within 0.058 sec.

From Fig. 4 all the time of 37 experiments after injected is greater than 3.1 and T = 2 sec, this error comes from the tasks schedule of operate system. It doesn't affect our understanding and analysis of experiment results.

Sensitivity: From Fig. 1, it can be seen that an application is put into memory image based on different kinds of region. For example there are code region, date region, heap region and stack region. In this study, present research focuses on code region. The characteristic of injecting faults injected into this region is that it almost causes application exception immediately. We can extract #3 and #10 from experiment records and plot them in Fig. 5.

In Fig. 5, value of y-axis comes from #10 of each experiment record. It only has two values: 0 and 1.1 means application failure and 0 no failure. X-axis comes from #3 of each experiment record. It is obvious that the failures are converge at the top third of injected code region, so we can conclude that this part of region is more sensitive to SEEs faults. The meaning of this parameter is that it will be easier to target the locations region where failure occurs once an application fails. Although, in this study we focus on injecting faults into code region, MOFI can inject faults into any regions; next we discuss how to compute sensitivity about different regions by records. For different applications the sizes of regions allocated are different from each other. It is hard to define sensitivity in this situation but for each region the corresponding sensitivity can be computed according to the way we compute sensitivity of code region discussed above. No matter what situation the method to compute the number of failures occurring in corresponding region is important. For example, code-num stands for number of failures occurring in code region. The value can be computed by Eq. 4.

$$\text{Code\_num} = \sum_{i=1}^{N} (\text{Inject\_addr}(i) \in \text{code\_region}) \times \text{f\_flag (i)} \tag{4}$$

where, Inject-addr is the #3 of each record and f-flag (i) is the #10. By Eq. 4 other numbers such as data-num, heap-num and stack-num can be computed. When we use uniform distribution to produce X (N) over all four regions, four numbers can be compared to analyze the sensitivity.

EXPERIMENTS BASED ON DIFFERENT X (N)

In above injecting experiment implemented by IMA, we use uniform random distribution to produce locations list X (N) over the whole code region. What about other distributions? Here, we will discuss the influences of the different distributions on experiments results.

We choose uniform, normal and exponential as three different distributions. Three different lists are produced by corresponding distributions, respectively on the same area of code region (Note that in this section the code region is different from the one used to demonstrate how to compute parameters in above section. The differences come from different original locations and different sizes of injected areas in code region). Then three campaigns are run. Each campaign includes N = 50 experiments as before and uses its own random distribution to generate list X (N). We use X-Exponential, X-Normal and X-Uniform to indicate corresponding experiment curves. In all three

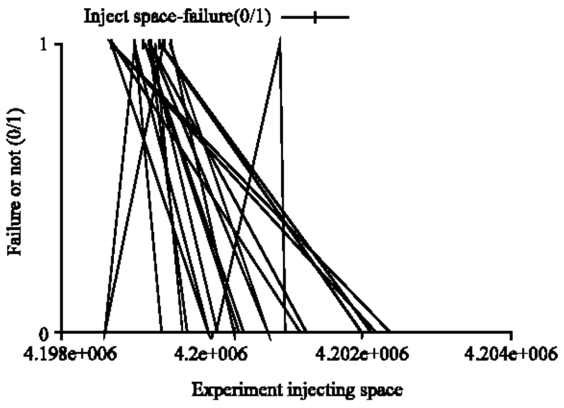

Fig. 5: Compute sensitivity

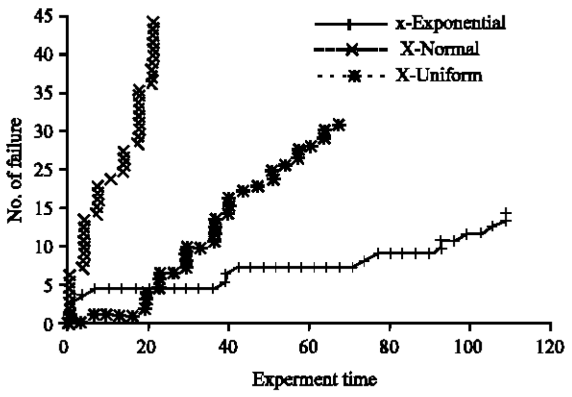

Fig. 6: Experiment time-failure number

campaigns T is set to 2 sec. Figure 6 shows experiment time and failure number of three campaigns, respectively.

In Fig. 6, x-axis stands for time of experiments executed. Y-axis stands for failure number of 50 experiments. There are three curves in this Fig. and each of them represents a distribution. It is obvious that when we use exponential random distribution to generate locations list X (N), corresponding curve indicated by X-Exponential, the time of 50 experiments is longest and when use normal random distribution, corresponding curve indicated by X-Normal, the time is shortest.

Figure 7 shows experiment number and failure number of three campaigns, respectively. X-axis stands for the number of experiments. It corresponds to #1 of each experiment. Y-axis is same with Fig. 6.

In Fig. 7 it can be found that when use normal random distribution the number of failures caused by injection is the most but exponential random is smallest. Comparing Fig. 6 and 7 a conclusion can be obtained, i.e., the more number of failures the less time of experiments. Now let's analyze the reason. By IMA set the time of finishing N experiments is t-all. Set t-obser = T×(N-fn) that is time injector need to observe application when injection doesn't make it

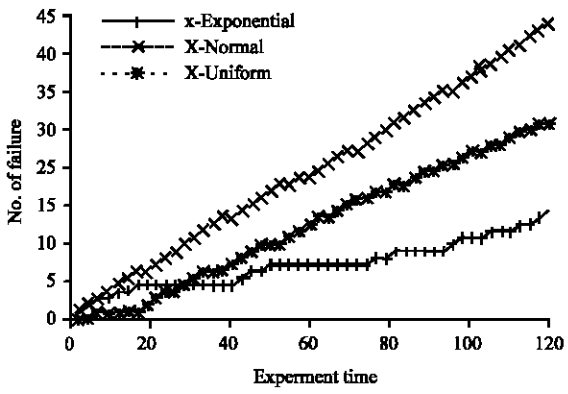

Fig. 7: Experiment number- failure number

failure. Set t-fail = MTTF×fn that is whole time needed to observe application on condition that it fails during all N experiments. So,

$$t\_all = t\_obser + t\_fail = T\times(N\_fn) + MTTF\times fn \tag{5}$$

where, T, N and MTTF are constant, then

$$dt\_all/dfn = -T + MTTF \tag{6}$$

Since, T = MTTF, dt_all/dfn = 0. So, if fn is smaller, t_all is longer.

There are many other distributions that can generate X(N). Although, we don't test all of them, by Eq. 5 and 6 and Fig. 6 and 7 we could conclude that there always is a distribution that makes fn smallest and t-all longest.

If one kind of distribution generating X(N) can make more failures than others with same number of experiments, it will be considered better than others. This is because the more failures make more problems found. These problems can be used as clues to deploy fault-tolerant mechanism in order to improve the reliability of application and system. It can be seen that in Fig. 6 and 7 X-Normal is a better distribution than others in the three campaigns.

CONCLUSION

A fault injection tool MOFI for use in injecting experiments on Windows platform has been developed. Using this tool reliability parameters of application were computed and analyzed. In the course of the experiments, the different location lists X(N) generated by different random distributions were used to execute injecting experiments. By analysis of experiments result, it proves that there always exists a better distribution than others to run injecting experiment by MOFI.

ACKNOWLEDGMENT

This research is supported by the Aeronautical Innovation Fund under grants No.CASC0409, China.

REFERENCES

Baldini, A., A. Benso, S. Chiusano and P. Prinetto, 2000. BOND: An interposition agents based fault injector for windows NT. Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Oct. 25-27, Yamanashi, Japan, pp: 387-395.

Blanc, S., J. Gracia and P.J. Gil, 2002. A fault hypothesis study on the TTP/C using VHDL-based and pin-level fault injection techniques. Proceedings of the 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, Nov. 06-08, Vancouver, BC., Canada, pp: 254-262.

Carreira, J., H. Madeira and J.G. Silva, 1998. Xception: A technique for the experimental evaluation of dependability in modern computers. IEEE Trans. Software Eng., 24: 125-136.

Dawson, S., F. Jahamian and T. Mitton, 1996. ORCHESTRA: A probing and fault injection environment for testing protocol implementations. Proceedings of the 2nd International Computer Performance and Dependability Symposium, Sept. 04- 06, Urbana-Champaign, pp: 56-56.

Gunnels, J.A., R.A. van-de-Geijn, D.S. Katz and E.S. Quintana-Ortí, 2001. Fault-tolerant high-performance matrix multiplication: Theory and practice. Proceedings of the 2001 International Conference on Dependable Systems and Networks, July 01 - 04, Goteborg, Sweden, pp: 47-56.

Han, S., K.G. Shin and H.A. Rosenberg, 1995. DOCTOR: An integrated software fault injection environment for distributed real-time systems. Proceedings of the International Computer Performance and Dependability Symposium, Apr. 24-26, Erlangen, Germany, pp: 204-213.

Kanawati, G.A., N.A. Kanawati and J.A. Abraham, 1995. FERRARI: A flexible software-based fault and error injection system. IEEE Trans. Comput., 44: 248-260.

Madeira, H., M. Rela, F. Moreira and J.G. Silva, 1994. RIFLE: A general purpose pin-level fault injector. Lecture Notes Comput. Sci., 852: 199-216.

McLoughlin, I.V., V. Gupta, G.S. Sandhu, S. Lim and T.R. Bretschneider, 2003. Fault tolerance through redundant COTS components for satellite processing applications. Proceedings of the 2003 Joint Conference of the Fourth International Conference on Information, Communications and Signal Processing and the Fourth Pacific Rim Conference on Multimedia, Dec. 15-18, Tait Electron. Ltd., Christchurch, New Zealand, pp: 296-299.

Oh, N., P.P. Shirvani and E.J. McCluskey, 2002. Control flow checking by software signatures. IEEE Trans. Reliability, 51: 111-122.

Some, R.R. and D.C. Ngo, 1999. REE: A COTS-based fault tolerant parallel processing supercomputer for spacecraft onboard scientific data analysis. Proceedings of the 18th Digital Avionics Systems Conference, Oct. 24, Pasadena, CA., pp: 3-12.

Some, R.R., W.S. Kim, G. Khanoyan, L. Callum, A. Agrawal and J.J. Beahan, 2001. A software-implemented fault injection methodology for design and validation of system fault tolerance. Proceedings of the 2001 International Conference on Dependable Systems and Networks, July 1-4, Pasadena, CA. USA., pp: 501-506.

Stott, D., P.H. Jones, M. Hamman, Z. Kalbarczyk and R.K. Iyer, 2002. NFTAPE: Networked fault tolerance and performance evaluator. Proceedings of the International Conference on Dependable Systems and Networks, June 23-26, Washington, DC., USA., pp: 542-543.

Torres-Pomales, W., 2000. Software fault tolerance: A tutorial. Technical Report: NASA-2000-tm210616.